計算機アーキテクチャ 86－3
（1991 1．24）
Appl. No. 10/815,742
Doc. Ref.: **AR11**

DSN型スーパースカラ・プロセッサ・プロトタイプの分岐パイプライン

原 哲也 久我守弘 村上和彰 富田眞治
（九州大学大学院総合理工学研究科）

我々は現在，DSNS (Dynamically-hazard-resolved,Statically-code-scheduled,Nonuniform Superscalar) アーキテクチャに基づくスーパースカラ・プロセッサの開発を行っている．スーパースカラ・プロセッサにおいては，分岐命令およびそれに起因する制御依存の存在によって，命令フェッチの阻害，後続命令実行の阻害，分岐遅延によるパイプラインの乱れ，および，命令レベル並列性の低下といった分岐ペナルティが生じ，性能が著しく低下する．このような分岐ペナルティの影響を緩和するため，DSNSプロセッサでは，①静的分岐予測＋分岐先バッファ，②投機的命令実行，③先行条件決定方式，および，④早期分岐解消といった手法を用いた分岐アーキテクチャを採用している．

本稿では，分岐アーキテクチャ，および，分岐命令のパイプライン処理過程ついて述べる．

Branch Pipeline of the DSNS Processor Prototype (in Japanese)

Tetsuya HARA, Morihiro KUGA, Kazuaki MURAKAMI, and Shinji TOMITA
Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyusyu University
6-1 Kasuga-koen, Kasuga-shi, Fukuoka, 816 Japan
e-mail : hara@is.kyushu-u.ac.jp

A DSNS (Dynamically-hazard-resolved,Statically-code-scheduled,Nonuniform Superscalar) processor prototype, has been being built at Kyushu University.

Control hazards due to branches cause a severe performance loss for superscalar processors. The DSNS processor prototype alleviates these effects with ①static branch prediction with branch-target-buffer, ②speculaitve execution, ③advanced conditioning, and ④early branch resolution.

This paper presents the branch architecture and the branch pipeline of the DSNS processor prototype.

BEST AVAILABLE COPY

1.はじめに

我々は，命令レベルの並列性を活用するプロセッサ・アーキテクチャとしてスーパースカラ方式を採用したプロセッサを開発中である．本プロセッサは以下のような特徴を持つ [3].

(1) 動的ハザード解消

命令間依存関係および資源競合に起因するハザードを実行時にハードウェアで検出し，解消する．これにより，オブジェクト・コードの互換性を保証する．

(2) 静的コード・スケジューリング

資源の競合や命令間の依存関係が存在すると，同時に実行できる命令数は減り，命令レベル並列性が減少する．パイプラインが滞ることなく流れるためには，命令（発行）順序の並べ換え（コード・スケジューリング）を行い同時に実行できる命令数を増す必要がある．このコード・スケジューリングをコンパイル時に静的に行う．これにより並列性の増大を図ると同時に，動的コード・スケジューリングにより生じるハードウェアの増大という問題を回避している．

(3) 非均質型機能ユニット

命令フェッチ機構，デコード機構，レジスタ・ポート，バスなどは，スーパースカラの多重度に従って多重化しているが，機能ユニットについては必要なものだけを多重化している．使用頻度の高い機能ユニットのみを多重化しているので，コスト／パフォーマンスが良い．

上記の特徴に基づき，現在開発中のプロセッサを DSNS (Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscalar) プロセッサと呼んでいる．

さて，命令パイプライン・プロセッサにおいては，分岐命令およびそれに起因する制御依存の存在によって性能が低下する．スーパースカラ・プロセッサにおいては，これら分岐ペナルティの影響が著しい．分岐ペナルティの主な原因には以下のものがある．

①命令フェッチの阻害：分岐するか否か（条件分岐の場合），および，分岐先アドレスが確定するまで次にフェッチすべき後続命令が決まらない．

②制御依存による後続命令実行の阻害：もし次にフェッチすべき後続命令を予測してフェッチしても，制御依存が解消するまでこれら後続命令の実行は開始あるいは終了できない．これは，制御依存関係にある命令（つまり，実行するか否かが未定の命令）がレジスタ内容等を更新してしまうと正確なマシン状態が保証できないことによる．

③分岐遅延によるパイプラインの乱れ：しかも，分岐予測が外れた場合は，パイプラインをフラッシュして正しい命令を再フェッチしなければならない．よって，分岐命令実行の遅延時間が長くなると，それだけパイプラインの乱れは増大する．

④命令ミスアラインメント [1] [16]：スーパースカラ・プロセッサでのみ問題となる．すなわち，分岐先アドレスが命令フェッチ・バウンダリに一致していない場合，フェッチした命令中に無駄な命令が含まれる．これにより，命令レベル並列性が低下する．

上記①②③の分岐ペナルティの影響を緩和するために，DSNS プロセッサでは以下の手法を採用している．

ⓐ静的分岐予測＋分岐先バッファ
ⓑ投機的命令実行
ⓒ先行条件決定方式
ⓓ早期分岐解消

なお，上記④の命令ミスアラインメントに関しては，ハードウェアによる対処も可能ではあるが [1]，DSNS プロセッサでは全面的にコンパイラに頼っている．

本稿では，分岐ペナルティに対する対処法（2章）を整理した後，DSNS プロセッサで採用した分岐アーキテクチャ（3章），DSNS プロセッサの概要（4章），および，分岐命令のパイプライン処理過程（5章）について述べる．

2. 分岐ペナルティへの一般的な対処法

2.1 分岐命令への対処

分岐命令の存在により生じる分岐ペナルティの影響を軽減する方法として，以下の多くのものが提案されている．なお，下記 (1) ～ (4) の方法はほぼ直交関係にあり，実現に際しては様々な組合せが可能である．

(1) 分岐方式

まずは，通常の compare-and-branch 方式や branch-on condition 方式とは異なる方式として，次のものがある．

①先行条件決定方式 (advanced conditioning) [2] [12]：条件決定（分岐するか否か）を分岐に先行して行い，分岐命令実行に伴う分岐遅延の低減を図る．

②分岐予告方式 [9]：分岐予告命令（prepare-to-branch instruction）により分岐先命令を予めフェッチしてバッファリングする．これにより，分岐命令が TAKEN の場合のパイプラインの乱れを抑える．

(2) 分岐命令

分岐命令の機能として，以下のものを加える．

①遅延分岐 [11]：分岐命令の遅延スロットを定義し，遅延スロット内の命令は制御依存を受けないようにする．つまり，分岐結果（TAKEN／NOT-TAKEN）に関係なく必ず実行される．もし，遅延スロットをすべて埋めることが出来たら，分岐ペナルティは生じない．しかし，制御依存を受けないような命令を発見するのは難しい．そこで，分岐結果によっては遅延スロット内の命令を無効化するような無効化付遅延分岐（delayed branch with squashing）も提案されている [10].

②静的分岐予測 [10]：コンパイラが意味解析結果およびプロファイル情報に基づいて，個々の条件分岐命令について分岐予測を行う．予測結果は，OP コードに反映される．これにより，命令フェッチが阻害されるのを防ぐ．

(3) 命令フェッチ

命令フェッチを中断することなく連続して行う方法として，以下のものがある．

①動的分岐予測 [8] [13]：静的分岐予測とは異なり，ハードウェアが実行時に分岐予測を行う．これにはさらに，次の手法がある．

ⓐnot-taken予測：not-taken（分岐しない）と予測して，非分岐先の命令流をフェッチし続ける．
ⓑtaken予測：taken（分岐する）と予測して，分岐先の命令流をフェッチするようにする．
ⓒ分岐予測バッファ（BPB：Branch Prediction Buffer）：個々の分岐命令の実行時の履歴をバッファリングしておき，それに基づいて分岐予測する．
ⓓ分岐先バッファ（BTB：Branch Target Buffer）[13]：BPB にさらに分岐先アドレスまたは分岐先命令をバッファリングする．これにより，taken 予測の場合，アドレス計算を行わずに直ちに分岐先命令の供給が可能となる．

②複数命令流フェッチ [8] [9]：非分岐先と分岐先の両方の命令流をフェッチする．

(4) 分岐命令実行

分岐命令の実行に当り，次のような配慮を行う．

①早期分岐解消 (early branch resolution) [9]：命令パイプラインの早いステージで分岐命令を実行する．これにより，制御依存の存在時間，および，分岐命令の実行遅延時間の低減を図る．

②分岐命令畳込み (branch folding) [6]：通常の命令パイプ

ラインの前に命令プリフェッチ・ステージを設け，無条件分岐命令はそこで実行を済ませる．よって，命令パイプラインには無条件分岐命令は入って行かない．しかし，条件分岐命令は命令パイプラインで実行される．無条件分岐命令に関しては，分岐ペナルティは生じない．

2.2 制御依存への対処

分岐命令の後続命令は一般的に，当該分岐命令に対して制御依存関係にある．すなわち，分岐命令の後続命令をフェッチできたとしてもこれらを実行するか否かは，制御依存が解消するまで定まらない．したがって，このような制御依存関係にある命令の命令パイプライン内での処置については，特別な配慮が必要である．これに関しては，少なくとも以下の2方法がある．

(1) 非投機的実行

制御依存が解消するまで，当該制御依存関係にある命令の実行を開始しない．実現は容易だが，分岐命令を越えた命令の実行が出来ない．つまり，最大同時に実行可能な命令が，1個の基本ブロックに限定される．よって，基本ブロック内の命令数が少ない場合，命令レベル並列性が落ちる．

(2) 投機的実行 (speculative execution)

制御依存が解消しなくても，当該制御依存関係にある命令の実行を開始する．ただし，制御依存が解消しないうちは，これらの命令の実行結果がレジスタ内容等を変更することを禁止する．もし，制御依存が解消した結果これらの命令の実行が不要と判明した場合は，命令自身および実行結果を無効化しなければならない．よって，実現は後述するようにやや複雑になる．しかし，分岐命令を越えた命令の実行が可能となり，命令レベル並列性の向上が期待できる．

さて，投機的実行の具体的な実現方式には，次の2つがある．

①条件付実行モード (conditional mode)：分岐予測によりフェッチされた命令は，対応する分岐命令に起因する制御依存が解消するまで条件付実行モード下に置かれる．条件付実行モード下にある命令は実行を行うことは可能だが，実行結果でレジスタ内容等を更新することは出来ない．

②ブースティング (boosting) [15]：個々の命令に対して，条件付実行モード下に置くか否かをコンパイラが決定する．条件付実行モード下に置かれた命令をブースト (boost) 命令と呼び，その指定はOPコードにより行う．ブースト命令はオブジェクト・コード上，対応する分岐命令より前に位置する．すなわち，上記①とは逆に，条件付実行モード下の命令が対応する分岐命令より先にフェッチされ実行を開始する．よって，静的コード・スケジューリング次第では，上記①より投機的実行の効果が期待できる．

また，条件付実行モード下にある命令 (ブースト命令も含む) が制御依存の解消を待たずにレジスタ内容等を変更するのを防ぐ手段として，以下のものがある．

ⓐバッファ方式 [14]：レジスタ・ファイルにバッファを設ける．このバッファの使用方法には，次の2種類がある．

(i) reorder buffer [16] [14]：条件付実行モード下にある命令の実行結果をバッファリングする．そして，制御依存が解消した時点で分岐予測が当たっていれば，その内容をレジスタに反映する．ただし，本バッファからもオペランド・フェッチが出来るよう，バイパス機構が必要となる．

(ii) history buffer [14]：条件付実行モード下にある命令の実行結果でレジスタ内容を変更するのを許す．ただし，そのとき元のレジスタ内容を本バッファにバッファリングする．そして，制御依存が解消した時点で分岐予測が外れていれば，その内容でレジスタ内容を復元する．

ⓑfuture file [14]：同一レジスタ・ファイルを2組設け，一方をアーキテクチャ上定義されたレジスタ・ファイル (architectural file)，他方を条件付実行モード下にある命令の実行結果書込み用ファイル (future file) とする．そして，制御依存が解消した時点で分岐予測が当たっていれば，future fileの内容をarchitectural fileに反映する．この反映方法には，ファイル間転送による方法 [14] [15]，および，3.2節で述べるレジスタ・アクセスパスの切り換えによる方法 [4] がある．

ⓒcheckpoint repair [7]：制御依存が発生した時点 (チェックポイント) におけるレジスタ内容を保存しておく．そして，当該制御依存が解消した時点で分岐予測が外れていれば，その内容でレジスタ内容を復元する．

3. 分岐アーキテクチャ

2章で挙げた対処法のうち，DSNSプロセッサでは以下の手法を採用している．

①静的分岐予測＋分岐先バッファ
②投機的実行：条件付実行モードおよびブースティング
③先行条件決定方式
④早期分岐解消

3.1 静的分岐予測＋分岐先バッファ

従来は動的分岐予測と組み合せて用いていた分岐先バッファ (BTB) を，DSNSプロセッサでは静的分岐予測と組み合わせる．

コンパイラは静的分岐予測結果に基づいて，個々の分岐命令に対して以下のいずれかの型を指定する．

①BTB登録型：taken予測，かつ，分岐先アドレスが不変であると判断された分岐命令はこの型になる．実行時に生成した分岐先アドレスをBTBに登録する．

②BTB非登録型：not-taken予測，あるいは，taken予測だが分岐先アドレスが変化すると判断された分岐命令はこの型になる．分岐先アドレスはBTBに登録されない．

BTBに分岐先アドレスが登録されると，対応する分岐命令はtakenと予測され，その登録分岐先アドレスが次サイクルの命令フェッチに使用される．

BTBの構成については4.1節で述べる．また，BTBへの分岐先アドレスの登録処理，および，BTBを用いた命令フェッチ処理については5.2節で述べる．

3.2 投機的実行

投機的実行方式として，2.2節で述べた条件付実行モードおよびブースティングの双方を採用する．両方式のハードウェア上の実現方法は基本的に等価であり，条件付実行モードのためのハードウェア機構がブースティングにも流用できる．

さて，条件付実行モード下にある命令 (ブースト命令も含む) が制御依存の解消を待たずにレジスタ内容等を変更するのを防ぐ手段として，DSNSプロセッサでは多重化レジスタ・ファイル [4] を採用している．これは，future file [14] の1実現方法である．ここで，多重化レジスタ・ファイルの多重度は，「条件付実行モードのレベル＋1」となる．DSNSプロセッサの条件付実行モードのレベル数は1であるので，多重度は2となる (3.4節参照)．これから，2重化レジスタ・ファイル (DRF：Dual Register File) と呼ぶ．

2重化レジスタ・ファイルにおいては，各レジスタは論理的には1個であるが，物理的には2個の実体 (物理レジスタ) を有する．そして，一時には，一方がカレント (current) 状態，他方がオルタネート (alternate) 状態となる．オルタネート状態にはさらに，有効／無効の2状態が存在する．また，カレント状態およびオルタネート状態にある物理レジスタを便宜上，カレント・レジスタおよびオルタネート・レジスタとそれぞれ呼ぶ．

2重化レジスタ・ファイルの動作の概要を以下に述べる．

(1) レジスタ読出し

無条件実行モード (unconditional mode：制御依存関係に

ない状態）下の命令は，そのソース・オペランドをカレント・レジスタから読み出す．一方，条件付実行モード下にある命令は，そのソース・オペランドを次のようにして得る．

①オルタネート・レジスタが有効な場合：オルタネート・レジスタから読み出す．

②オルタネート・レジスタが無効な場合：カレント・レジスタから読み出す．

(2) レジスタ書込み

無条件実行モードで終了した命令の実行結果は，カレント・レジスタに書き込む．一方，条件付実行モードで終了した命令の実行結果は，オルタネート・レジスタに書込み当該オルタネート・レジスタを有効状態とする．

(3) 分岐命令の実行終了

制御依存が解消されると，個々のレジスタについて次の状態遷移が起きる．

①分岐予測が当たった場合：有効なオルタネート・レジスタがカレント状態となり，対を成すカレント・レジスタはオルタネート無効状態になる．それ以外のレジスタは，状態に変化なし．

②分岐予測が外れた場合：すべてのオルタネート・レジスタは無効化状態となる．カレント・レジスタは，状態に変化なし．

3.3 先行条件決定方式

3.3.1 従来の条件分岐方式

条件分岐は，一般に以下の処理から成る．

①コンディション生成：条件分岐の元になるコンディション（状態）を生成する．このコンディションは，算術・論理演算等を実行することにより生成される．

②条件決定（conditioning）：①で生成したコンディションを分岐条件でテストし，分岐するか否かを決定する．

上記①②は分岐するしないに関わらず行う．分岐する場合は，さらに以下の処理が必要である．

③分岐先アドレス生成：分岐先の命令アドレスをアドレッシング・モードに従って計算する．なお，アドレッシング・モード次第で，この処理は省ける．

④分岐処理：③で生成した分岐先アドレスをプログラム・カウンタ（PC）に設定する．

さて，従来の一般的な条件分岐方式には，以下の2つがある．

(1) compare-and-branch 方式

表1に示すように，条件分岐処理過程の①②③④のすべてを1個の compare-and-branch 命令で行う．クリティカルパスが①→②→④と長いため分岐遅延が長くなる欠点がある．

(2) branch-on-condition 方式

コンディション・コード（CC：Condition Code）を用いて，①と②③④とを別々の命令で行う．表1に示すように，①は通常の算術・論理演算命令における CC 設定で，②③④は1個の branch-on-condition 命令で行う．branch-on-condition 命令自身のクリティカルパスが②→④ないし③→④と短くなるため，compare-and-branch 命令に比べて分岐遅延は短い．しかし，CC を導入したことで，CC に関するフロー依存関係の検出，および，CC へのアクセス競合回避といった課題が生じる．

3.3.2 先行条件決定方式

表1．　条件分岐方式

処理 / 方式	①コンディション生成	②条件決定	③分岐先アドレス生成	④分岐
compare-and-branch	compare-and-branch 命令			
branch-on-condition	算術・論理演算命令	branch-on-condition 命令		
advanced-conditioning	算術・論理演算命令	test 命令	branch 命令	
	compare-and-test 命令			

前項で述べた条件分岐方式以外に，図1に示す先行条件決定方式（advanced-conditioning）がある［12］［2］．DSNS プロセッサでは本方式を採用する．本方式は表1に示すように，①は branch-on-condition 方式同様，通常の算術・論理演算命令における CC 設定で行うが，②と③④は別々の命令で行う．③④は1個の branch 命令で行い，そのクリティカルパスは③→④と短い．

先行条件決定方式の導入に関連して，次の2種類のレジスタを定義する．

ⓐTF レジスタ（TFR：True／False Register）：32個の1ビット長レジスタで，「分岐が成立するか（True＝TAKEN）／否か（False＝NOT-TAKEN）」を保持する．

ⓑタグ付き汎用／浮動小数点レジスタ：個々の汎用／浮動小数点レジスタに4ビットのタグを付ける．算術・論理演算命令の実行により生成された CC は，そのデスティネーション・レジスタのタグに格納される（図1参照）．これにより，従来の branch-on-condition 方式における CC に関する課題を解決している［2］．

また，関連命令として，次の3種類の命令を定義する（図1参照）．

ⓐテスト（test）命令：ソースレジスタのタグ内の CC に対して分岐条件と合致するか否かのテストを行い，分岐する／しないの true／false 値をデスティネーション TF レジスタに設定する．条件分岐処理過程の②に相当する．

ⓑ比較＆テスト（compare-and-test）命令：2つのソースオペランド間の算術・論理比較を行い，その結果に対して分岐条件と合致するか否かのテストを行い，true／false 値をデスティネーション TF レジスタに設定する．条件分岐処理過程の①②に相当する．

ⓒ分岐（branch）命令：ソース TF レジスタの true／false 値に

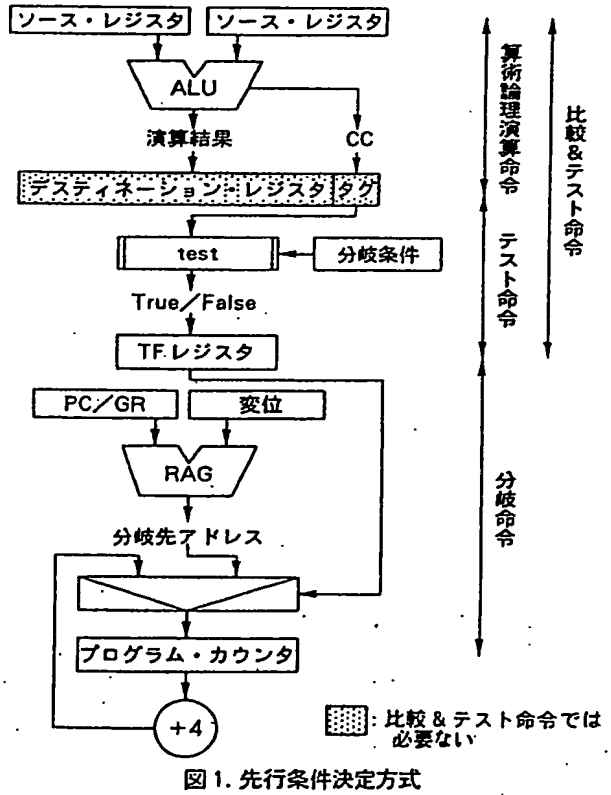


図1．先行条件決定方式

従って分岐結果（TAKEN／NOT-TAKEN）を決定する．TAKENの場合さらに，プログラム・カウンタ（PC）に分岐先アドレスを設定する．分岐先アドレスはアドレッシング・モード（PC相対／GR相対／PC+GR）に従って計算する．条件分岐処理過程の③④に相当する．

これ以外に，多方向分岐用に，2個のTFレジスタ間の論理演算を行う命令も用意している．

先行条件決定方式により，以下の利点が期待できる．

- テスト命令または比較＆テスト命令と分岐命令との間の距離を大きくようにスケジューリングすることで，条件決定（分岐するか否か）に伴う分岐遅延を隠蔽できる．
- TFレジスタ間の論理演算命令を用いることによって，複数の分岐命令を1個の分岐命令にまとめることができる．すなわち，分岐命令数の削減が可能である．
- 分岐命令で行う処理（TFR参照と分岐先アドレス生成）が少ないことから，3.4節で述べるように早期分岐解消が可能となる．その結果，分岐遅延自体を低減できる．

3.4 早期分岐解消

3.1-3.3節で述べた方法に加えて，分岐命令の実行自身の高速化，つまり，分岐の早期解消が重要である．これは，以下の理由に依る．

- 分岐予測が外れた場合，条件付実行モード下のすべての命令を無効化することで命令パイプラインを復元する（パイプライン・フラッシュ）．したがって，当該分岐命令に起因する制御依存の解消が遅くなるほど，多数の命令が無効化されることになる．当然，分岐ペナルティも増大する．
- 条件付実行モード下で新たな分岐命令をフェッチした場合，1レベルの条件付実行モードでは当該分岐命令ならびに後続命令の実行は開始できない．これは，条件付実行モードのレベルが足りないことに原因がある．しかし，条件付実行モードのレベル数の増加は，多重化レジスタ・ファイル（3.2節参照）の多重度の増加を意味し，ハードウェア量の増大に直接結びつく．よって，条件付実行モードのレベル数を2以上に上げるのは難しい．

このような背景から，DSNSプロセッサには早期分岐解消（early branch resolution）を採用している．5.2節で述べるように，分岐命令以外の一般命令は命令パイプラインのE（実行）ステージで実行を行うのに対して，分岐命令はそれよりも前のD（解読）ステージで実行を開始する．

しかし，このように一般命令とは異なるステージで分岐命令の実行を行うためには，余分なハードウェアが必要となるという問題が生じる．ところが，DSNSプロセッサでは，3.3節で述べたように，分岐命令で行うべき処理がTFR参照と分岐先アドレス生成だけと少ない．よって，必要とされるハードウェア量もさほど多くはない．この分岐命令専門のハードウェアは，5.1節で述べる分岐ユニットとして設けている．

3.5 分岐命令の仕様

DSNSプロセッサの分岐命令の仕様は以下の通りである．

(1) 種類

3.1節で述べた通り，BTB登録型とBTB非登録型の2種類に分類される．なお，すべての分岐命令は条件分岐命令である．分岐するか否かは，ソース・オペランドであるTFRの値によって決定される．TFR0の値は常にTure（=TAKEN）であり，これをソース・オペランドに指定する分岐命令が無条件分岐命令となる．

(2) アドレッシング・モード

分岐先アドレス生成の際のアドレッシング・モードとして，以下の3種類を用意する．

①PC（プログラム・カウンタ）＋オフセット（符号付き19ビット）
②GR（汎用レジスタ）＋オフセット（符号付き14ビット）
③PC+GR

なお，命令長が4バイトであるため，命令アドレスの下位2ビットは常に'00'である．よって，分岐先アドレス計算では上位30ビットのみを生成する．

4. プロセッサの概要

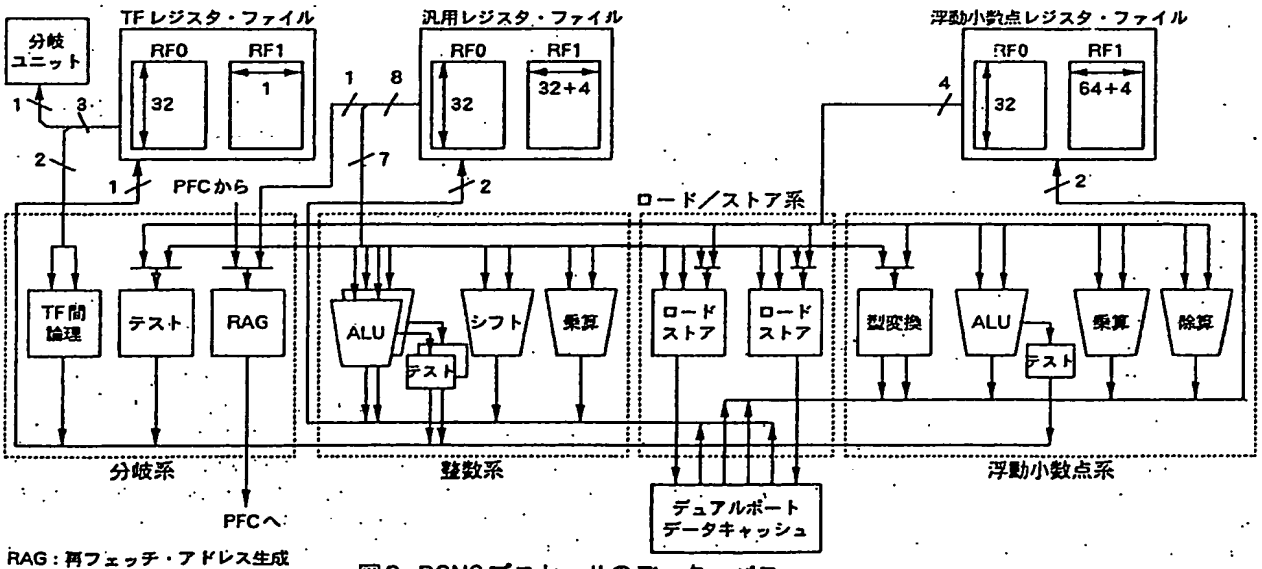
DSNSアーキテクチャの有効性を検証するため，DSNSアーキテクチャに基づく試作プロセッサ"DSNSプロセッサ"の開発を行っている．この章では，DSNSプロセッサの構成と命令パイプライン処理過程について，その概要を述べる［4］．

4.1 DSNSプロセッサの構成

DSNSプロセッサは，以下の主要ユニットから構成される．

(1) 命令キャッシュ（IC：Instruction Cache）

ポート・サイズ16バイトで，4バイト長命令4個から成る命令ブロックを一時にフェッチする．ライン・サイズ64バイト（=4



RAG：再フェッチ・アドレス生成

図2. DSNSプロセッサのデータ・パス

命令ブロック）のダイレクト・マッピング方式の仮想アドレス・キャッシュである．

命令ブロック毎に1個の予測分岐先アドレスを保持する分岐先バッファ（BTB：Branch Target Buffer）を備える．これにより，分岐命令の有無に関わらず，命令ブロックの連続フェッチが可能となる．BTBと　命令キャッシュは，タグ・アレイを共用する．

(2) プリデコーダ（PD：PreDecoder)

フェッチした4命令をプリデコードして，コンフリクト・チェックに必要な情報を得る．また，分岐命令実行に関するすべての情報もここで生成する．

(3) デコーダ（D：Decoder)

各命令の実行制御に必要な情報，特に機能ユニットの制御情報をROMから読み出す．

(4) コンフリクト・チェッカ（CC：Conflict Checker)

プリデコード結果に基づいて，最大4命令に対して，
・命令ブロック内の命令間のフロー依存および出力依存の検出
・先行命令ブロックに対するフロー依存および出力依存の検出
・機能ユニット競合の検出と調停
・レジスタ・ファイルの読出しポートの割当て
を行う．

(5) 分岐ユニット（BU：Branch Unit)

早期分岐解消を行うための分岐命令実行専用ユニットで，3段のパイプライン構成となっている．また，分岐命令専用のコンフリクト・チェッカ（BCC）と，逐次的に分岐命令を実行していくための4段の分岐命令バッファ（BIB：Branch Instruction Buffer）を備える．詳細は5章で述べる．

(6) 機能ユニット（FUs：Functional Units)

図2に示すように，以下の4系統，計13個の機能ユニットを備える．最大4命令が毎サイクル，任意の機能ユニットの組合せに対して発行可能である．

①整数系機能ユニット：ALU（×2），シフタ，乗算器
②浮動小数点系機能ユニット：ALU，乗算器，除算器，型変換器
③ロード／ストア機能ユニット：2つの独立したロード／ストア・ユニット［5］
④分岐系機能ユニット：再フェッチ・アドレス生成器，先行条件決定方式のための2つのユニット

(7) 2重化レジスタ・ファイル（DRFs：Dual Register Files)

図2に示すように，①汎用レジスタ（GR），②浮動小数点レジスタ（FPR），③TFレジスタ（TFR）の3系統のレジスタ・ファイルを備える．

(8) デュアルポート・データキャッシュ（DPDC：Dual-Port Data-Cache）［5］

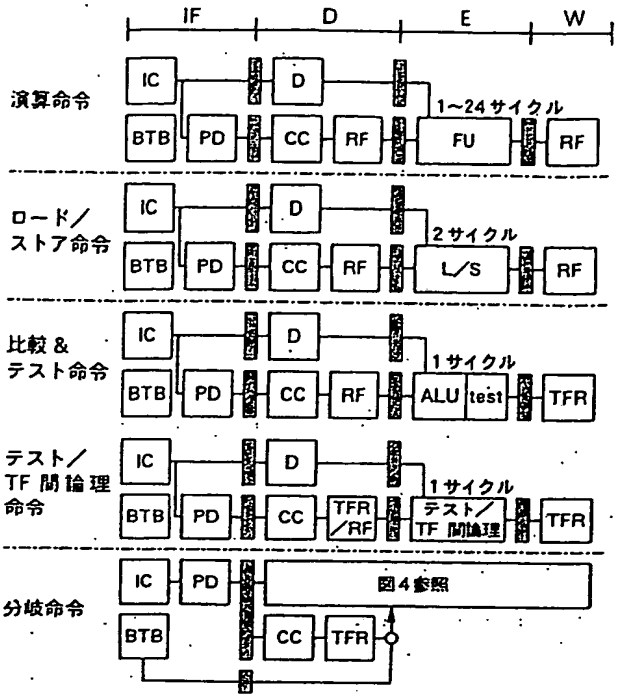
整数データおよび浮動小数点データを処理できる2重化したロード／ストア・パイプラインに対応するため，デュアルポート化している．ポート・サイズ8バイト，ダイレクト・マッピング方式の仮想アドレス・キャッシュである．

また，ミスヒットを起こしても後続のロード／ストア命令を受け付け可能な，ノンブロッキング・キャッシュとなっている．

4.2 命令パイプライン処理過程

命令パイプラインは，
①IF：命令ブロック・フェッチ＋プリデコード
②D：コンフリクト・チェック＋デコード＋オペランド・フェッチ
③E：実行
④W：書込み
の4ステージ構成である．パイプライン・サイクル時間は，60nsを目標にしている．

図3に，命令の種類毎に命令パイプライン処理過程を示す．分岐命令の処理については，5章で述べる．



CC：コンフリクト・チェッカ　　L/S：ロード／ストア・ユニット
D：デコーダ　　PD：プリデコーダ
FU：機能ユニット　　RF：レジスタ・ファイル
IC：命令キャッシュ　　TFR：TF レジスタ・ファイル

図3. パイプライン処理過程

5. 分岐パイプライン

分岐命令の処理は，IFステージにおける分岐予測，条件付実行モードの設定のあと，分岐パイプラインによって行われる．分岐パイプラインの実行主体は分岐ユニットであり，これはBD，BE，BWの3ステージから構成される．

5.1 分岐ユニット

分岐ユニット（BU：Branch Unit）には，以下の2個のユニット，および，分岐パイプラインのための制御機構を備える．

(1) 分岐命令バッファ（BIB：Branch Instruction Buffer)

分岐命令は基本的に逐次実行しなければならないので，分岐ユニットは一時に高々1つの分岐処理しか行わない．したがって，同一命令ブロックに複数の分岐命令が存在する場合に備えて，分岐ユニットは4命令を保持する分岐命令バッファ（BIB：Branch Instruction Buffer）を設けている．分岐ユニットでは，分岐命令を含む命令ブロックを，一旦BIBにバッファリングしたあと，分岐命令を1命令づつ順番に実行してゆく．

(2) 分岐命令用コンフリクトチェッカ（BCC：Branch Conflict Checker)

メインパイプラインのDステージとBDステージが同期している場合，分岐ユニットで処理中の分岐命令は，メインパイプラインのDステージにも存在する．よって，データ依存の検出にはメインパイプラインのCCを用いる．しかし，分岐ユニットとメインパイプラインの処理は独立しているため，フロー依存でBDステージがインターロックされる場合，分岐命令の所属する命令ブロックがEステージ以降に進み，Dステージには後続命令ブロックが存在する状態が生じる可能性がある．つまり，当該分岐命令は，一般命令用のDステージには存在せず，CCを使用することができない．このため，分岐ユニットには，分岐命令専用のコンフリクトチェッカ（BCC：Branch Conflict

Checker）を備え，上記のようにCCが使用できない場合のデータ依存検出を行う．BCCは，一般命令のCC同様，先行命令ブロックに対するデータ依存の検出を行うが，命令ブロック内の命令間データ依存の検出は行わない．
また，機能ユニット（再フェッチ・アドレス生成器）およびレジスタ・ファイル読出しポートは分岐ユニット専用となっているので，

・機能ユニット競合の検出と調停
・レジスタ・ファイルの読出しポートの割当て

を行う必要がない．

5.2 分岐命令処理過程

5.2.1 IFステージ

IFステージは，分岐命令以外の一般命令と共通のステージである．分岐に関しては，以下の処理を行い，その結果をメインパイプラインのDステージと分岐パイプラインのBDステージへ送る．

(1) 分岐予測

PFC（プリフェッチ・カウンタ）値による命令キャッシュからの4命令（命令ブロック）のプリフェッチと同時に，対応するBTBエントリの読み出しを行う．そのエントリが有効であれば当該命令ブロック内に存在する分岐命令をtakenと予測する．この場合，次サイクルのプリフェッチはBTBエントリに登録された分岐先アドレスを用いて行われる．

(2) NOP置換

分岐予測がtakenの場合，当該命令ブロックに存在し，分岐予測に対応する分岐命令の後続命令をすべてNOPとする．

(3) 制御依存検出とインターロック

プリデコードの結果，分岐命令ないしブースト命令の有無が判明すると，以下のように条件付実行モードを設定，および，インターロック制御を行う．

①分岐パイプラインに実行未終了の分岐命令が存在する場合：

(i) 命令ブロック内に分岐命令が存在する場合：分岐パイプライン中の分岐命令が実行終了するまで，IFステージをインターロックする．実行終了したら，②の処理を行う．

(ii) 命令ブロック内に分岐命令が存在しない場合：すべての命令をレベル1の条件付実行モードとして，Dステージへと進める．

②分岐パイプラインに実行未終了の分岐命令が存在しない場合：

(i) 命令ブロック内に分岐命令が存在する場合：分岐命令より後の命令をすべて条件付実行モードとする．複数の分岐命令が同一命令ブロックに存在する場合，条件付実行モードのレベルを分岐命令数に比例して増やす．すなわち，1番目の分岐命令以降はレベル1，2番目の分岐命令以降はレベル2，となる．レベル2以上の条件付実行モード下の命令が存在しても，IFステージにおいてはこれらの命令のインターロックは行わない．しかし，これらの命令は，Dステージにおいて命令の発行をブロックされる．

(ii) 分岐命令が存在しない場合：命令ブロック内の命令はすべて，無条件実行モードとなる．

ブースト命令については，上記①②で設定される条件付実行モードをさらに1レベル上げて設定する．

5.2.2 分岐パイプライン処理過程

分岐パイプラインにおける処理は，以下の要因によって支配される．

ⓐ分岐予測（taken／not-taken）
ⓑアドレッシング・モード（GRアクセスが必要／不要）
ⓒ分岐命令の型（BTB登録型／非登録型）
ⓓ分岐結果（TAKEN／NOT-TAKEN）

上記ⓐⓑⓒはIFステージで，ⓓはBDステージで判明する．

図4に分岐パイプラインの処理の概要を示す．

(1) 制御依存解消

データ依存検出の結果，当該分岐命令のソース・オペランドであるTFRにフロー依存がない場合，TFRを読み出す．読み出したTF値が分岐結果となる．これと予測結果を比較し，分岐予測の当否を判定する．

ⓐ分岐予測が当たった場合（図4のケース①②⑤⑥⑩）：すべての命令の条件付実行モードを1レベル下げる．つまり，レベル1の条件付実行モードで実行されている命令は，無条件実行モードとし，レベル2の条件付実行モードの命令は，レベル1となる．また，2重化レジスタ・ファイルに関しては，3.2節で述べたように，有効なオルタネート・レジスタをカレント・レ

分岐予測	アドレッシング・モード	分岐結果	型	分岐パイプライン（BD / BE / BW）	ケース	分岐ペナルティ（サイクル）
not-taken	PC相対	NOT-TAKEN	非登録型	RAG (TA)	①	0
			登録型	RAG (TA) — BTB	②	0
		TAKEN	非登録型	RAG (TA) — RIF (TA)	③	1
			登録型	RAG (TA) — RIF (TA)／BTB	④	1
	GR相対／PC+GR	NOT-TAKEN	非登録型	CC — GRF	⑤	0
			登録型	CC — GRF — RAG (TA) — BTB	⑥	0
		TAKEN	非登録型	CC — GRF — RAG (TA) — RIF (TA)	⑦	2
			登録型	CC — GRF — RAG (TA) — RIF (TA)／BTB	⑧	2
taken	—	NOT-TAKEN	登録型	RAG (SA) — RIF (SA)	⑨	1
		TAKEN		RAG (SA)	⑩	0

ⓐ分岐予測
ⓑアドレッシング・モード
ⓒ型
ⓓ分岐結果

BTB：BTB登録
CC：コンフリクト　チェック
RAG：再フェッチ・アドレス生成
GRF：汎用レジスタ・ファイル
RIF：命令再フェッチ
TA：分岐先アドレス

図4.　分岐パイプライン処理過程

ジスタに切り換える．図4のケース①②⑥⑩はBDステージで，ケース⑥はBEステージでそれぞれステージのインターロックを解除する．

ⓑ分岐予測が外れた場合（図4のケース③④⑦⑧⑨）：すべての条件付実行モードの命令を無効化する．また，オルタネート・レジスタも無効化する．

(2) 再フェッチ・アドレス生成

分岐予測が外れた際に必要となる再フェッチ・アドレスの生成を行う．再フェッチ・アドレスは，機能ユニットの1つであるRAG (Re-fetch Address Generator) を用いて計算する．再フェッチアドレス生成処理は，以下のように，予測結果とアドレッシング・モードによって支配される．

ⓐ分岐予測がnot-takenの場合：再フェッチアドレスとして，分岐先アドレスを生成する．

(i) アドレッシング・モードがPC相対の場合（図4のケース①④）：BDステージでアドレス計算を行う．

(ii) アドレッシング・モードがGR相対およびPC+GRの場合（図4のケース⑤⑧）：BDステージではGRへのアクセスを行い，BEステージでアドレス計算を行う．

ⓑ分岐予測がtakenの場合（図4のケース⑨⑩）：再フェッチアドレスとして，非分岐先アドレスを生成する．GRへのアクセスが不要なので，常にBDステージでアドレス計算を行う．また，BTB登録型の分岐命令で，まだBTBに登録されていない命令（すなわちnot-taken予測された命令）は，分岐結果（TAKEN／NOT-TAKEN）に関わらず，分岐先アドレスをBTBに登録する（図4のケース②④⑥⑧）．

(3) パイプライン復元処理

分岐予測が外れた場合，誤った予測によりフェッチされた命令およびその実行結果を無効化（パイプライン・フラッシュ）すると同時に，再フェッチ・アドレスを用いて，正しい命令流を再フェッチ（RIF：Re-Instruction Fetch）する（図4のケース③④⑦⑧⑨）．

5.3 分岐ペナルティ

分岐命令のソース・オペランドに対するフロー依存はないと仮定したときの，分岐命令に起因するパイプラインの乱れ（分岐ペナルティ）は以下のようになる．

ⓐ分岐予測ヒットの場合（図4のケース①②⑤⑥⑩）：ペナルティなし．

ⓑ・PC相対で分岐予測ミス，あるいは，taken予測で分岐予測ミスの場合（図4のケース③④⑨）：1サイクルのペナルティ．

ⓒ・GR相対またはPC+GR，かつ，not-taken予測で分岐予測ミスの場合（図4のケース⑦⑧）：2サイクルのペナルティとなる．

制御依存は，TFRのフロー依存がなければ，すべてBDステージで解消される．

6. おわりに

以上，DSNSプロセッサの，分岐アーキテクチャ，および，分岐パイプライン処理について述べた．

分岐ペナルティの影響を軽減するために，以下の手法を採用している．

①静的分岐予測＋分岐先バッファ
②投機的実行：条件付実行モードおよびブースティング
③先行条件決定方式
④早期分岐解消

現在，ハードウェア開発を進めると同時に，ソフトウェア・シミュレータを開発している．採用した上記の4手法は互いに密接に関係し合っているので，これらの間の相互作用について今後評価を行う予定である．

また，本分岐アーキテクチャは，最適化コンパイラの存在を前提としている．静的分岐予測，ブースティング，および，先行条件決定方式については，最適化コンパイラ抜きにしては効果が期待できない．特に，ブースティングおよび先行条件決定方式を有効たらしめるには，高度な静的コード・スケジューリング・アルゴリズムの開発が必須である．これら最適化コンパイラ技術については，別の機会に報告したい．

参考文献

[1] 原ほか："SIMP（単一命令流／多重命令パイプライン）方式に基づくスーパースカラ・プロセッサ『新風』の命令供給機構，" 情処研報，ARC-80-7（1990年1月）．

[2] 原ほか："『新風』プロセッサの条件分岐方式，" 情処40回全大，7L-6（1990年3月）．

[3] 村上ほか："SIMP（単一命令流／多重命令パイプライン）方式に基づくスーパースカラ・プロセッサの改良方針，" 信学技報，CPSY-90-54（1990年7月）．

[4] 原ほか："SIMP（単一命令流／多重命令パイプライン）方式に基づく改良版スーパースカラ・プロセッサの構成と処理，" 信学技報，CPSY-90-55（1990年7月）．

[5] 納富ほか："DSN型スーパースカラ・プロセッサ・プロトタイプのロード／ストア・パイプライン，" 情処研報，ARC-86-4（1991年1月）．

[6] Ditzel,D.R. and McLellan,H.R.: "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proc. 14th Ann. Int'l. Symp. Computer Architecture, pp.2-9, June 1987.

[7] Hwu,W.W. and Patt,Y.N.: "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," IEEE Trans. Comput., vol.C-36, no.12, pp.1496-1514, Dec. 1987.

[8] Lee,J.K.F. and Smith,A.J.: "Branch Prediction Strategies and Branch Target Buffer Design," Computer, vol.17, no.1, pp.6-22, Jan. 1984.

[9] Lilja,D.J.: "Reducing the Branch Penalty in Pipelined Processors," Computer, vol.21, no.7, pp.47-55, July 1988.

[10] McFarling,S. and Hennessy,J.: "Reducing the Cost of Branches," Proc. 13th Ann. Int'l. Symp. Computer Architecture, pp.396-403, June 1986.

[11] Pleszkun,A.R. et al.: "WISQ: A Restartable Architecture Using Queues," Proc. 14th Ann. Int'l. Symp. Computer Architecture, pp.290-299, June 1987.

[12] Rosocha,W.G. and Lee,E.S.: "Performance Enhancement of SISD Processors," Proc. 6th Ann. Symp. Computer Architecture, pp.216-231, Apr. 1979.

[13] Smith,J.E.: "A Study of Branch Prediction Strategies," Proc. 8th Ann. Symp. Computer Architecture, pp.135-148, May 1981.

[14] Smith,J.E. and Pleszkun,A.R.: "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans. Comput., vol.37, no.5, pp.562-573, May 1988.

[15] Smith,M.D.,Lam,M.S., and Horowitz,M.A.: "Boosting Beyond Static Scheduling in a Superscalar Processor," Proc. 17th Ann. Int'l. Symp. Computer Architecture, pp.344-354, May 1990.

[16] Sohi,G.S. and Vajapeyam,S.: "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," Proc. 14th Ann. Int'l. Symp. Computer Architecture, pp.27-34, June 1987.

Computer Architecture 86 – 3
(January 24, 1991)

Branch Pipeline of DSN Superscalar Processor Prototype

Tetsuya HARA, Morihiro KUGA, Kazuaki MURAKAMI and Sinji TOMITA
(Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University)

At present, we are developing a superscalar processor based upon DSNS (Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscalar) architecture. In the superscalar processor, branch instructions and the existence of control dependence, which result from the branch instructions, may cause branch penalties, such as interference in an instruction fetch, interference in the execution of successor instructions, disarray of a pipeline due to a branch delay, and the reduction of the parallelism of an instruction level. These result in a remarkable reduction of performance. In order to alleviate the influence of branch penalties, in the DSNS processor, techniques using branch architecture, such as ① static branch prediction with a branch-target-buffer, ② speculative instruction execution, ③ advanced conditioning and ④ early branch resolution, is adopted.

In the present paper, the branch architecture and the pipeline processes of the branch instructions are described.

Branch Pipeline of the DSNS Processor Prototype (in Japanese)

Tetsuya HARA, Morihiro KUGA, Kazuaki MURAKAMI and Shinji TOMITA
Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
6 – 1 Kasuga-koen, Kasuga-shi, Fukuoka 816, Japan
e-mail: hara@is.kyushu-u.ac.jp

A DSNS (Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscalar) processor prototype has been built at Kyushu University.

Control hazards due to branches cause a severe performance loss for superscalar processors. The DSNS processor prototype alleviates these effects with ① static branch prediction with branch-target-buffer, ② speculative execution, ③ advanced conditioning and ④ early branch resolution.

This paper presents the branch architecture and the branch pipeline of the DSNS processor prototype.

1. Introduction

We are developing a processor, where a superscalar method has been adopted as processor architecture where the parallelism of an instruction level is utilized. This processor has the following characteristics [3]:

(1) Dynamic hazard resolution:

Hazards which result from the dependency between instructions and a resource competition, are detected by hardware, and are resolved. This resolution results in the guarantee of the compatibility of an object code.

(2) Static code scheduling:

If the resource competition or/and the dependency between instructions exist, the number of instructions, which can be executed at the same time, decreases, resulting in the reduction of the parallelism of an instruction level. For the purpose of permitting a pipeline flow without stagnating, it is necessary to rearrange the (issuing) order of instructions (code scheduling) and to increase the number of instructions that can be simultaneously executed. This code scheduling is statically performed upon compiling, and attempts to increase parallelism. At the same time, a hardware increasing problem, caused by the dynamic code scheduling, is avoided.

(3) Heterogeneous functional units:

An instruction fetch mechanism, a decoding mechanism, a register port and path, etc. are multiplexed by following the superscalar multiplicity. However, only essential functional units are multiplexed. Because only the functional units that are frequently used are multiplexed, cost performance is excellent.

Based upon the above-mentioned characteristics, the processor, currently being developed is referred to as a DSNS (Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscalar) processor.

In an instruction pipeline processor, its performance decreases due to branch instructions and the existence of control dependence, which results from the branch instructions. In a superscalar processor, the influence of branch penalties is remarkable. The primary causes of the branch penalties are as follows:

① Interference in instruction fetch: a successor instruction(s), which should be fetched next, cannot be determined until deciding whether or not branching is performed (in the case of the conditional branching) and until a branch target address becomes definite.

② Interference in the execution of successor instructions due to control dependence: If the successor instruction which should be fetched next is predicted and fetched, the execution of the successor instruction cannot be started and completed. This is because if instructions in the control dependency (in other words, instructions for which it has not yet been decided whether they should be executed), update the register contents etc., a precise machine state cannot be guaranteed.

③ Disarray of pipeline due to branch delay: In addition, when the branch prediction is incorrect, the pipeline has to be flashed and a correct instruction has to be re-fetched. Therefore, the

longer the delay time for the execution of the branch instruction becomes, the more the disarray of the pipeline worsens.

④ Instruction misalignment [1] [15]: This is a problem only in the superscalar processor. In other words, when the branch target address does not match an instruction fetch boundary, an unnecessary instruction is included in the fetched instruction, reducing the parallelism of an instruction level.

For the purpose of alleviating the influence of branch penalties in the above ①, ② and ③, in the DSNS processor, the following techniques have been adopted:

(a) Static branch prediction with a branch-target-buffer;
(b) Speculative instruction execution;
(c) Advanced conditioning; and,
(d) Early branch resolution.

Furthermore, concerning instruction misalignment in the above ④, it is possible to manage with hardware [1]. However, the DSNS processor entirely relies upon a compiler.

In the present paper, after Action on Branch Penalties (Chapter 2) is organized, Branch Architecture (Chapter 3) adopted in the DSNS processor, Outline of DSNS Processor (Chapter 4) and Pipeline Process of Branch Instructions (Chapter 5) are described.

2. General Action on Branch Penalties:

2.1 Managing Branch Penalties:

As a method for reducing the influence of branch penalties which may occur due to the existence of branch instructions, many methods have been proposed as follows. Furthermore, the methods in the below-mentioned (1) through (4) have a substantially orthogonal relationship, and when realized, various combinations are possible.

(1) Branch method:

At first, as a method different from a normal compare-and-branch method or a branch-on-condition method, there are the following methods:

① Advanced conditioning (advanced conditioning) [2] [12]: conditioning (whether or not branching is performed) is performed in advance of branching, and attempts to reduce branch delays accompanied with the execution of branch instruction(s).

② Prepare-to-branch method [9]: A branch target instruction is fetched in advance according to a prepare-to-branch instruction (prepare-to-branch instruction), and is buffered. This results in the restraint of the disarray of the pipeline when the branch instruction is 'TAKEN'.

(2) Branch instructions:

As functions for the branch instructions, the following are added:

① Delayed branch [11]: A delay slot of the branch instruction is defined, designed such that an instruction(s) within the delay slot is not affected by control dependence. In other words, regardless of the branch result (TAKEN/ NOT-TAKEN), the instruction(s) is always executed. If the delay slot can be completely filled

in, branch penalties will not occur. However, it is difficult to discover an instruction that will not be affected by control dependence. Then, a delayed branch with quashing (delayed branch with squashing) is also proposed where an instruction(s) within a delay slot will be quashed depending upon the branch result [10].

② Static branch prediction [10]: A compiler performs the branch prediction relative to each conditional branching instruction, based upon a meaning analysis result(s) and profile information. The prediction results reflect an OP code, preventing interference in the instruction fetch.

(3) Instruction fetch:

Methods to perform the instruction fetch in succession are as follows:

① Dynamic branch prediction [8] [13]: Differing from the static branch prediction, the branch prediction is performed when hardware is in operation. There are additionally the following techniques in this method:

(a) 'Not-taken' prediction: Predicting 'not-taken' results in the successive fetch of an instruction flow in a non-branch target.
(b) 'Taken' prediction: It is designed such that a prediction 'taken' shall result in fetching an instruction flow in a branch target.
(c) Branch prediction buffer (BPB: Branch Prediction Buffer): A history upon the execution of each branch instruction is buffered, and the branch prediction is performed based upon the history.
(d) Branch target buffer (BTB: Branch Target Buffer) [13]: A branch target address or a branch target instruction is additionally buffered to the BPB, enabling the immediate provision of the branch target instruction without address calculation, when 'Taken' is predicted.

② Fetch of multiple instruction flows [8] [9]: Both of instruction flows in a non-branch target and a branch target are fetched.

(4) Execution of Branch Instruction:

For the execution of a branch instruction, the following care-and-attention is performed:

① Early branch resolution (early branch resolution) [9]: A branch instruction is executed in the early stage of the instruction pipeline. This attempts to shorten the existence period of the control dependence and the execution time delay of the branch instruction.

② Branch instruction folding (branch folding) [6]: An instruction pre-fetch stage is established before a normal instruction pipeline, and any unconditional branch instruction shall be executed there. Therefore, the unconditional branch instruction will never enter into the instruction pipeline. However, a conditional branch instruction shall be executed in the instruction pipeline. Concerning the unconditional branch instruction, no branch penalty will occur.

2.2 Management to control dependence:

Successor instructions of a branch instruction are generally in control dependency with the branch instruction. In other words, even if the successor instructions of the branch instruction can be fetched, whether or not these are executed will not be determined until the control dependence is resolved. Therefore, concerning the processing of the instructions in the control dependency within an instruction pipeline, special consideration is required, in regard to which there are at least the following two methods:

(1) Nonspeculative execution:

The execution of the instructions, which are in the control dependency, shall not start until the control dependence is resolved. Its realization is easy. However, no instruction above that of the branch instruction can be executed. In other words, simultaneously executable maximized instructions are limited to one basic block. Therefore, when the number of instructions within a basic block is small, the parallelism of the instruction level is reduced.

(2) Speculative execution:

Even though the control dependence is not resolved, the execution of instructions, in the control dependency, is started. It is prohibited to update register contents, etc. with the execution results of the instructions as long as the control dependence is not resolved. If it turns out to be unnecessary to execute the instructions as a result of the control dependence resolution, the instructions themselves and the execution results have to be quashed. Therefore, its realization becomes slightly complicated as mentioned below. However, instructions above the branch instruction become executable, and the improvement of the parallelism of the instruction level can be anticipated.

There are two specific realization methods of the speculative execution, as follows:

① Conditional execution mode (conditional mode): Instruction(s), which was fetched according to a branch prediction, is placed in the conditional execution mode until the control dependence, which results from its correspondent branch instruction, is resolved. It is possible to execute any instructions in the conditional execution mode. However, the register contents, etc. cannot be updated with the execution results.

② Boosting (boosting) [16]: A compiler determines whether or not each instruction is placed in the conditional execution mode. The instructions in the conditional execution mode are referred to as 'boost instructions,' and their designations are performed by the OP code. The boost instructions are positioned above the object code and before their correspondent branch instructions. In other words, contrary to the above-mentioned mode ①, the instructions in the conditional execution mode are fetched before their correspondent branch instructions, and execution is started. Therefore, depending upon static code scheduling, the efficacy of the speculative execution can be anticipated according to the above-mentioned mode ①.

Further, as a means for preventing the update of the register contents, etc. by the instructions in the conditional execution

mode (including the boost instructions) without waiting for the resolution of the control dependence, there are the following means:

(a) Buffer method [14]: A buffer is established in a register file. For how to use this buffer, there are the following two types:
  (i) Recorder buffer [16] [14]: Execution results of instructions, which are in the conditional execution mode, are buffered. Then, when the control dependence is resolved, if the branch prediction is correct, the contents are incorporated to the register, and a bypass mechanism is required to be able to fetch an operand from the buffer.
  (ii) History buffer [14]: This allows the register contents to be updated with the execution results of instructions in the conditional execution mode, on which occasion the original register contents must be buffered in ths history buffer. When the control dependence is resolved, if the branch prediction is incorrect, the register contents are restored by the original contents buffered in the history buffer.

(b) Future file [14]: Two sets of the same register files are established, one of which is regarded as a register file defined in terms of the architecture (architectural file), and the other being regarded as a file where execution results of instructions in the conditional execution mode, are written (future file). When the control dependence is resolved, if the branch prediction is correct, the contents in the future file will be incorporated in the architectural file. For this incorporation method, there are a method by a transfer between files [14] [15], and another method by switching a register access path [4], described in Section 3.2.

(c) Checkpoint repair [7]: Contents in a register at a point when the control dependence occurs (checkpoint) are stored. When the control dependence is resolved, if the branch prediction is incorrect, the contents in the register are restored with the [stored] contents.

3. Branch Architecture:

Among the actions mentioned in Chapter 2, in the DSNS processor, the following techniques have been adopted:

① Static branch prediction with a branch target buffer;
② Speculative execution: Conditional execution mode and boosting;
③ Advanced conditioning; and
④ Early branch resolution.

3.1 Static branch prediction with a branch target buffer:

In the DSBS processor, the branch target buffer (BTB) conventionally used in combination with a dynamic branch prediction is combined with static branch prediction.

A compiler designates either of the following types to each branch instruction, based upon the results of static branch prediction:

① BTB registration type: A branch instruction, where it is decided that its branch prediction is 'taken', and that its branch target address is fixed, is this type. The branch target address

generated upon execution, is registered in the BTB.

② BTB non-registration type: A branch instruction, where it is decided that its branch prediction is 'not-taken', or that the prediction is 'taken' but its branch target address will change, is this type. The branch target address is not registered to the BTB.

When the branch target address is registered to the BTB, its correspondent branch instruction is predicted as 'taken', and its registered branch target address is used for an instruction fetch in the next cycle.

The configuration of the BTB s described in Section 4.1. Further, the registration process of the branch target address to the BTB and the instruction fetch process using the BTB are described in Section 5.2.

3.2 Speculative execution

As a speculative execution method, both the conditional execution mode and the boosting described in Section 2.2, are adopted. The realization method of both methods on the hardware is basically equivalent, and the hardware mechanism for the conditional execution mode can also be diverted to that for boosting.

As a means of preventing the update of register contents, etc. by instructions (including boost instructions), placed in the conditional execution mode, without waiting for the resolution of control dependence in the DSNS processor, a multiplexed register file [4] has been adopted. This is one realization method for the future file [14]. Herein, the multiplicity of the multiplexed register file becomes the "level of the conditional execution mode + 1". The level of the conditional execution mode in the DSNS processor is '1', so the multiplicity becomes '2' (refer to Section 3.4). Hereafter, it is referred to as a 'dual register file (DRF: Dual Register File)'.

In the dual register file, there is logically one entity in each register. However, there are physically two entities (physical register). Temporarily, one becomes a current state, and the other becomes an alternate state. In the alternate state, additional two states, which are valid and invalid, exist. Further, the physical registers in the current state and in the alternate state are referred to as a 'current register' and an 'alternate register' as a matter of convenience, respectively.

The outline of operations of the dual register file is described as follows:

(1) Reading out from register:

An instruction placed in the unconditional execution mode (unconditional mode: state where this instruction is not in control dependency) reads out its source operand from a current register. In the meantime, an instruction placed in the conditional execution mode obtains its source operand as follows:

① When an alternate register is valid: Read out from the alternate register.

② When an alternate register is invalid: Read out from a current register.

(2) Writing into register:

An execution result(s) of an instruction, which was completed in the unconditional execution mode, is written into a current register; in the meantime, an execution result(s) of an instruction, which was completed in the conditional execution mode, is written into an

alternate register, and the alternate register is set as the valid state.

(3) Execution completion of branch instruction:

When the control dependence is resolved, the following state transition occurs to each register:

① When the branch prediction is correct: A valid alternate register becomes a current state, and a current register, which is a counterpart of the alternate register, becomes an invalid state. No states change in registers other than these two registers.
② When the branch prediction is incorrect: All alternate registers become an invalid state. No states change in current registers.

3.3 Advanced conditioning
3.3.1 Conventional conditional branching method:

Conditional branching is generally comprised of the following processes:

① Condition generation: A condition that causes the conditional branching is generated. The execution of arithmetic and logical operation, etc. results in the generation of this condition.
② Conditioning (conditioning): The condition, which was generated in the process ①, is tested under the branch condition, and whether or not branching is performed should be decided.

The above-mentioned processes ① and ② are performed regardless of whether or not branching is performed. In the case of branching is performed, the following processes are additionally required:

③ Branch target address generation: An instruction address for a branch target is calculated by following an addressing mode. Furthermore, this process can be omitted depending upon the addressing mode.
④ Branching process: The branch target address, which was generated in the process ③, is set to a program counter (PC).

In the meantime, there are the following two conventional general conditional branching methods:

(1) Compare-and-branch method:

As shown in Table 1, all of the conditional branching processes ①, ②, ③ and ④ are performed by one compare-and-branch instruction. Because its critical path is long, which is ① → ② → ④, there is a defect such that the branch delay becomes longer.

(2) Branch-on-condition method:
Using a condition code (CC:

Condition Code), the process ① and the processes ②, ③ & ④ are performed by separate instructions. As shown in Table 1, the process ① is performed by the CC setting in the normal arithmetic/ logical operation instruction, and the processes ②, ③ & ④ are performed by one branch-on-condition instruction. The critical path of the branch-on-condition instruction itself is shorter, which is ② → ④ or ③ → ④, so the branch delay is shorter compared to the compare-and-branch instruction. However, the introduction of the CC results in the creation of new problems, such as the detection of a flow dependency concerning the CC and the avoidance of access competition to the CC.

Table 1: Conditional branching method

Processing	①	②	③ Branch	④

<table>
<tr><th>Method</th><th>Condition generation</th><th>Conditioning</th><th>target address generation</th><th>Branching</th></tr>
<tr><td>Compare-and-branch</td><td colspan="4">Compare-and-branch instruction</td></tr>
<tr><td>Branch-on-condition</td><td>Arithmetic/ logic operation instruction</td><td colspan="3">Branch-on-condition instruction</td></tr>
<tr><td rowspan="2">Advanced conditioning</td><td>Arithmetic/ logic operation instruction</td><td>Test instruction</td><td colspan="2" rowspan="2">Branch instruction</td></tr>
<tr><td colspan="2">Compare-and-test instruction</td></tr>
</table>

3.3.2 Advanced conditioning

Other than the conditional branching methods described in the previous section, there is advanced conditioning (advanced conditioning) [12] [2]. In the DSNS processor, this method is adopted. In this method, as shown in Table 1, process ① is performed by the CC setting in the normal arithmetic/ logic operation instruction, as similar to the branch-on-condition method. However, process ② and processes ③ & ④ are performed by separate instructions. Processes ③ & ④ are performed by one branch instruction, and its critical path is short, which is ③ → ④.

In connection with the introduction of the advanced conditioning, the following two types of registers are defined:

(a) TF register (TFR: True/ False Register): This is a register that contains 32 of 1 bit length, and "whether or not branching is concluded (True = TAKEN)/ (False = NOT TAKEN)" is maintained.
(b) Tagged general-purpose/ floating point register: Each general-purpose/ floating point register is tagged with 4-bit. The CC, which was generated due to the execution of the arithmetic/ logic operation instruction, is stored in the tag of its destination register (refer to Fig. 1). This results in the resolution of the problem concerning the CC with the conventional branch-on-condition method [2].

Further, as a related instruction, the following three types of instructions are defined (refer to Fig. 1):

(a) Test (test) instruction: A test whether or not the branching condition is matched with the CC within a source register is conducted, and the true/false value for whether or not branching is performed is set to a destination TF register. This is equivalent to the process ② in the conditional branching processes.
(b) Compare-and-test (compare-and-test) instruction: The arithmetic and logic comparison between two source operands are conducted, and a test for whether or not the result is matched with the branching condition is conducted, and, a true/false value is set to the destination TF register. This is equivalent to the processes ① and ② in the conditional branching process.

(c) Branch (branch) instruction: A branch result (TAKEN/ NOT TAKEN) is decided by following a true/ false value in a source TF register. When the result is 'TAKEN', a branch target address is additionally set to the program counter (PC). The branch target address is calculated by following the addressing mode (PC relativity/ GR relativity/ PC + GR). This is equivalent to the processes ③ and ④ in the conditional branching process.

Other than these instructions, an instruction that performs a logic operation between two TF registers is also prepared for multidirectional branching.

According to advanced conditioning, the following advantages can be anticipated:

- Scheduling to have a great distance between the test instruction or the compare-and-test instruction and the branch instruction enables the concealment of the branch delay associated with the conditioning (whether or not branching is performed).
- Using the logic operation instruction between the TF registers enables the unification of multiple branch instructions into one branch instruction. In other words, it is possible to reduce the number of branch instructions.
- Because the processes (TFR reference and generation of branch target address) by the branch instruction are less, it becomes possible to resolve the branch in the early stage, which will

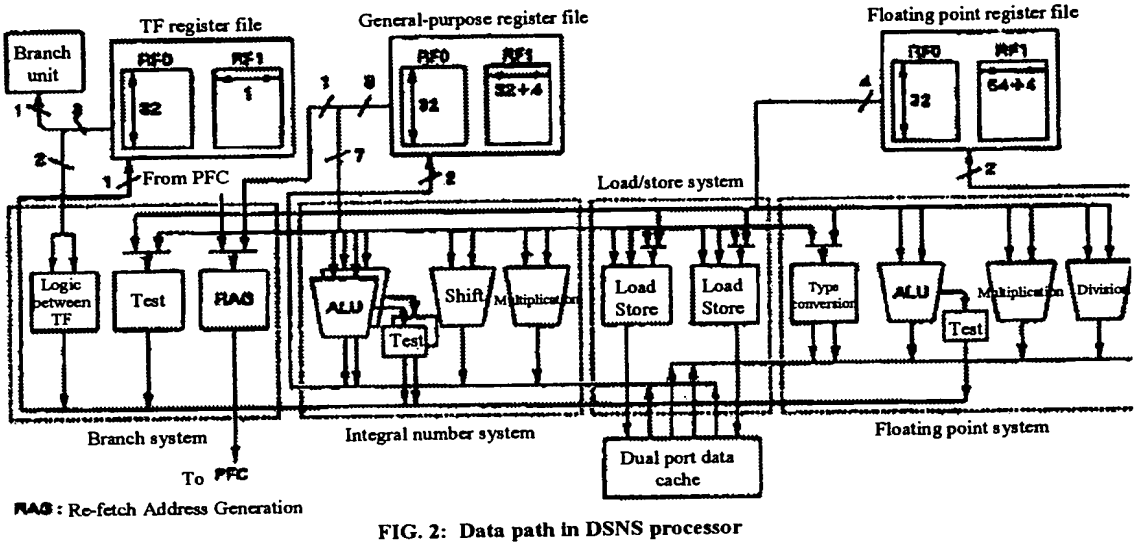


FIG. 2: Data path in DSNS processor

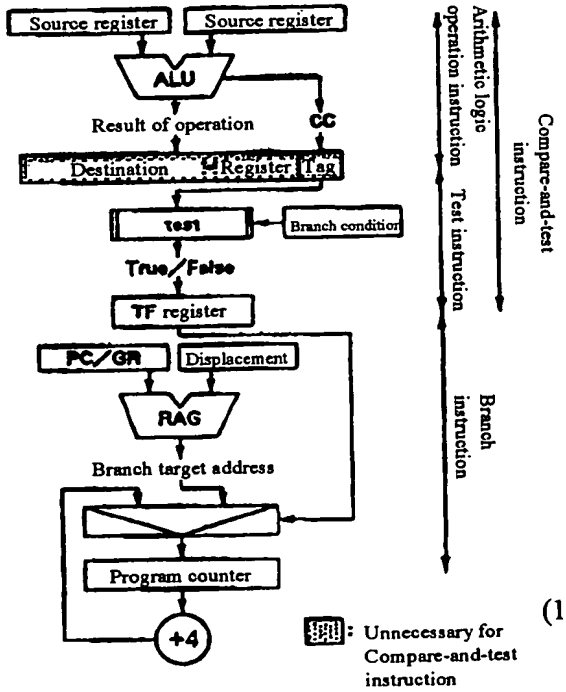


FIG. 1: advanced conditioning

(10)

be described in Section 3.4. As a result, the branch delay itself can be reduced.

3.4 Early branch resolution

In addition to the methods described in Sections 3.1 through 3.3, it is important to speed up the execution itself of a branch instruction; in other words, to resolve the branch in the early stage for the following reasons:

- When the branch prediction is incorrect, quashing all instructions, placed in the conditional execution mode, results in the restoration of the instruction pipeline (pipeline flash). Therefore, the later the control dependence, which results from the branch instruction, is resolved, the more instructions shall be quashed. Naturally, the branch penalties shall also increase.
- When a new branch instruction is fetched in the conditional execution mode, the execution of the branch instruction and its successor instruction(s) cannot be started in the level 1 conditional execution mode. This causes the lack of level in the conditional execution mode. However, the increase of the level in the conditional execution mode means the increase of the multiplicity of a multiplexed register files (refer to Section 3.2), and it is directly related to the increase of the hardware. Therefore, it is difficult to increase the level in the conditional execution mode to 2 or higher.

According to this background, in the DSNS processor, early branch resolution (early branch resolution) is adopted. As described in Section 5.2, general instructions other than branch instructions are executed in the E (execution) stage of the instruction pipeline; in the meantime, the execution of the branch instruction is started in the D (decoding) stage, which is previous to the E stage.

However, as explained above, for the purpose of executing branch instructions in the different stage from the stage where the general instructions are executed, the problem occurs where extra hardware is required. However, in the DSNS processor, as described in Section 3.3, processes which should be performed by branch instruction are less, which are only the TFR reference and the generation of a branch target address. Therefore, necessary hardware capacity is not that much. Hardware that is exclusively used for branch instructions is established as a branch unit, described in Section 5.1.

3.5 Specification of branch instructions

The specifications of the branch instructions in the DSNS process are as follows:

(1) Type:

As described in Section 3.1, the branch instructions are classified into two types, the BTB registration type and the BTB non-registration type. Furthermore, all branch instructions are conditional branching instructions. Whether or not branching is performed is decided according to a TFR value, which is a source operand. The TFR0 value is always True (= TAKEN), and the branch instructions that designate this value to the source operand become unconditional branch instructions.

(2) Addressing mode:

As an addressing mode when generating a branch target address, the following three types are prepared:

① PC (program counter) + offset (signed 19 bits)
② GR (general-purpose register) + offset (signed 14 bits)
③ PC + GR

Furthermore, because the instruction length is 4 bytes, low-order 2 bits in an instruction address is always '00'. Therefore, in the calculation for a branch target address, only the high-order 30 bits shall be generated.

4. Outline of processor

In order to verify the validity of the DSNS architecture, we have been developing a prototype processor, "DSNS processor", based upon the DSNS architecture. In this chapter, the outline of the configuration of the DSNS processor and the instruction pipeline processes shall be described [4].

4.1 Configuration of DSNS processor:

The DSNS processor is comprised of the following primary units:

(1) Instruction cache (IC: Instruction Cache):

The port size is 16 bytes, and an instruction block comprised of four 4-byte length instructions, is simultaneously fetched. This is a virtual address cache using a direct mapping method with 64 bytes (= four instruction blocks) of the line size.

Each instruction block is equipped with a branch target buffer (BTB: Branch Target Buffer) where one prediction branch target address is maintained. This enables the successive fetching of the instruction blocks regardless of the existence of branch instructions. The BTB and the instruction cache share a tag array.

(2) PreDecoder (PD: PreDecoder):

Four fetched instructions are pre-decoded, and information necessary for a conflict check, is obtained. Further, all information concerning the execution of branch instructions is also generated here.

(3) Decoder (D: Decoder):

Information necessary for the execution control of each instruction, especially, control information of the functional units is read out from a ROM.

(4) Conflict checker (CC: Conflict Checker):

Based upon the results of pre-decoding, the following are performed to four instructions at maximum:

- Detection of a flow dependence and an output dependence between instructions within an instruction block;
- Detection of a flow dependence and an output dependence relative to a successor instruction block;
- Detection and adjustment of functional unit competition; and
- Allocation of read-out report in a register file.

(5)Branch unit (BU: Branch Unit):

This is a unit exclusively used for the execution of branch instructions for the purpose of the early branch resolution, and it is constructed with three stages of pipelines. Further, it is equipped with a conflict checker that is exclusively used for branch instructions (BCC), and four stages of branch instruction buffers (BIB: Branch Instruction Buffer) for the purpose of successively executing branch instructions. The details are described in Chapter 5.

(6)Functional Units (FUs: Functional Units)

As shown in Fig. 2, it is equipped with the following four systems, a total of 13 functional units. Four instructions are issuable to a combination of optional functional units per cycle at maximum.

① Integral number functional units: ALU (×2), shifter, multiplier
② Floating point functional units: ALU, multiplier, divider, type converter
③ Load/store functional units: two independent load/store units [5]
④ Branch functional units: re-fetch address generator, two units for the advanced conditioning

(7)Dual register files (DRFs: Dual Register Files)

As shown in Fig. 2, it is equipped with three systems of register files, which are ① general-purpose register (GR), ② floating point register (FPR) and ③ TF register (TFR).

(8)Dual-port data-cache (DPDC: Dual-Port Data-Cache) [5]

In order to respond to a dual load/store pipeline where integral number data and floating number data can be processed, a dual-port is realized. The port size is 8 bytes, and it is a virtual address cache with a direct mapping method.

Further, it is a non-blocking cache where a successor load/store instruction(s) is receivable even if a 'mishit' occurs.

4.2 Instruction pipeline processes:

The instruction pipeline has a 4-stage construction as follows:

① IF: Instruction block fetch + pre-decode;
② D: Conflict check + decode + operand fetch;
③ E: Execution; and
④ W: Writing.

For the pipeline cycle time, we aim at 60 ns.

Fig. 3 shows the instruction pipeline processes per instruction type. The processing of the branch instructions shall be described in Chapter 5.

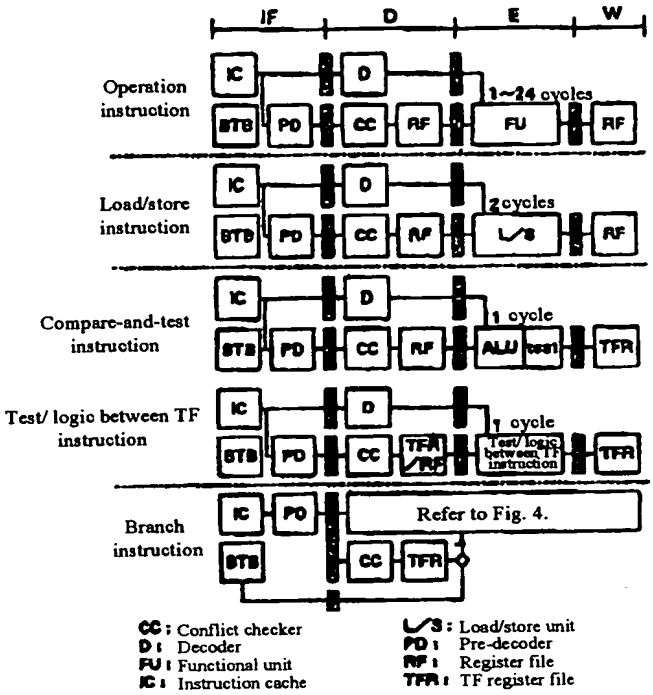


FIG. 3: Pipeline processes

5. Branch pipeline

The processing of the branch instructions is performed by the branch pipeline after the branch prediction in the IF stage and the setting of the conditional execution mode. The execution entity of the branch pipeline is a branch unit, comprised of three stages, BD, BE and BW.

5.1 Branch unit

The branch unit (BU: Branch Unit) is equipped with the following two units and a control mechanism for the branch pipeline.

(1) Branch instruction buffer (BIB: Branch Instruction Buffer)

Branch instructions basically have to be successively executed, so a branch unit performs no more than one branch processing at a given time. Therefore, in case multiple branch instructions exist in the same instruction block, the branch unit is provided with a branch instruction buffer (BIB: Branch Instruction Buffer) that maintains four instructions. In the branch unit, after an instruction block that includes branch instructions is buffered to the BIB once, the branch instructions are executed one by one in order.

(2) Conflict checker for branch instructions (BCC: Branch Conflict Checker)

When the D stage in a main pipeline and the BD stage are synchronized, the branch instructions being processed at the branch unit also exist in the D stage of the main pipeline. Therefore, for the detection of data dependence, the CC in the main pipeline is used. However, because the processing for the branch unit and the main pipeline is independent from each other, when the BD stage is interlocked due to the flow dependence, there is the possibility that the instruction block where the branch instruction belongs advances to the E stage or beyond, and that there is a situation where a successor instruction block(s) exists in the D stage. In other words, the branch instruction does not exist in the D stage, which is for general instructions, so the CC cannot be used. Consequently, the branch unit is equipped with a conflict checker exclusively used for branch instructions (BCC: Branch Conflict Checker), and the data dependence when the CC cannot be used as mentioned above is detected. The BCC detects data dependence relative to a precedent instruction block(s), similar to the CC for general instructions. However, it does not detect any data dependence between instructions within an instruction block.

Further, since the functional units (re-fetch/ address generator) and a read-out port in a register file are exclusively used for the branch unit, it is unnecessary to perform:

- Detection and adjustment of functional unit competition, and
- Allocation of read-out port in the register file.

5.2 Branch instructions processes

5.2.1 IF stage

An IF stage is a common stage with general instructions except for branch instructions. Concerning branching, the following processing is performed, and the results are transmitted to the D stage of the main pipeline and the BD stage of the branch pipeline.

(1) Branch prediction

Four instructions (instruction block) from an instruction cache are pre-fetched according to a PFC (pre-fetch counter) value; simultaneously, a correspondent BTB entry is read out. If the entry is valid, a branch instruction(s) that exists within the instruction block is predicted as 'TAKEN'. In this case, the pre-fetch for the next cycle is performed using a branch target address, registered in the BTB entry.

(2) NOP replacement

When the branch prediction is 'TAKEN', all successor instructions of branch instructions, which exist in the instruction block, and which correspond with the branch prediction, are regarded as NOP.

(3) Detection of control dependence and interlock

As a result of pre-decoding, if the existence of a branch instruction(s) or a boost instruction(s) is ascertained, the conditional execution mode is set and the interlock control is performed as follows:

① When a branch instruction, which has not yet completed the execution, exists in an instruction pipeline:
  (i) When a branch instruction exists within an instruction block: the IF stage is interlocked until the execution of the branch instruction in the branch pipeline is completed. When the execution is completed, the processing ② is performed.
  (ii) When no branch instruction exists within an instruction block: all instructions are placed in the level 1 conditional execution mode, and the processing progresses to the D stage.

② When no branch instruction, which has not yet completed the execution, exists in an instruction pipeline:
  (i) When a branch instruction exists within an instruction block: all instructions that are a successor of the branch instruction are placed in the conditional execution mode. When multiple branch instructions exist in the same instruction block, the level of the conditional execution mode is increased in proportion to the number of the branch instructions. In other words, the 1st branch instruction and its successor instructions will be set to level 1, and the 2nd branch instruction and its successor instructions will be set to level 2. Even if instructions, which are in the level 2 or higher conditional

execution mode, exist, in the IF stage, these instructions shall not be interlocked. However, the issue of these instructions shall be blocked in the D stage.

(ii) When no branch instruction exists [within an instruction block]: all instructions within an instruction block are placed in the unconditional execution mode.

Concerning boost instructions, the level of the conditional execution mode set in the above-mentioned cases ① and ②, is additionally increased by 1 and is set.

5.2.2 Branch pipeline processes

Processing in a branch pipeline is controlled by the following factors:

(a) Branch prediction (taken/ not-taken)
(b) Addressing mode (GR access is necessary/ unnecessary)
(c) Type of branch instruction (BTB registration type/ non-registration type)
(d) Branch result (TAKEN/ NOT-TAKEN)

The above-mentioned factors (a), (b) and (c) are ascertained in the IF stage, and the above-mentioned (d) is ascertained in the BD stage.

Fig. 4 shows the outline of processing of the branch pipeline.

(1) Resolution of control dependence

As a result of the detection of data dependence, when there is no flow dependence in a TFR, which is a source operand of the branch instruction, the TFR is read out. The read-out TF value is a branch result. Comparing this value to a prediction result, the propriety of the branch prediction is decided.

(a) When the branch prediction is correct (the cases ①, ②, ⑤, ⑥ and ⑩ in Fig. 4): the conditional execution mode of all instructions is reduced by 1 level. In other words, instructions that are executed in the level 1 conditional execution mode is placed in the unconditional execution mode, and instructions that are executed in the level 2 conditional execution mode is placed in the level 1 conditional execution mode. Further, concerning the dual register file, as described in Section 3.2, a valid alternate register is switched to a current register. In the cases ①, ②, ⑥ and ⑩ in Fig. 4, the interlock of the stage is released in the BD stage, and in the case ⑥ , it is

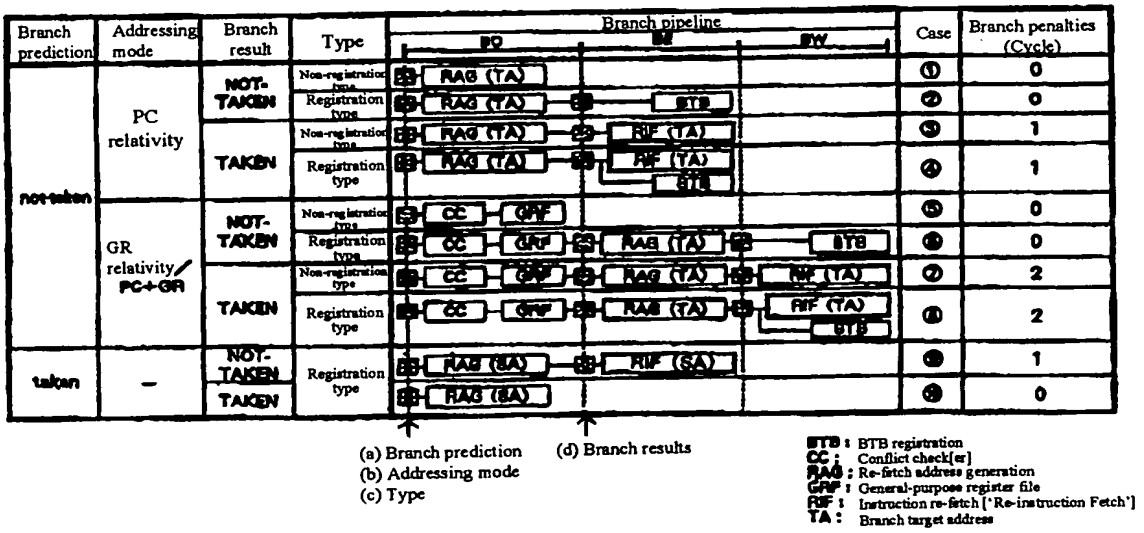


FIG. 4: Branch pipeline processes

released in the BE stage, respectively.

(b) When the branch prediction is incorrect (the cases ③, ④, ⑦, ⑧ and ⑨): all instructions that are executed in the conditional execution mode are quashed. Further, an alternate register(s) is also quashed.

(2) Re-fetch address generation

A re-fetch address, which becomes necessary on the occasion that a branch prediction is incorrect, is generated. The re-fetch address is calculated using an RAG (Re-fetch Address Generator), which is one of the functional units. The generation processing of the re-fetch address is managed by the prediction result and the addressing mode, as follows:

(a) When the branch prediction is 'not-taken': A branch target address is generated as a re-fetch address.
  (i) When the addressing mode is the PC relativity (the cases ① through ④ in Fig. 4): The address calculation is performed in the BD stage.
  (ii) When the addressing mode is the GR relativity and PC + GR (the cases ⑤ through ⑧): The access to the GR is performed in the BD stage, and the address calculation is performed in the BE stage.

(b) When the branch prediction is 'taken' (the cases ⑨ and ⑩): A non-branch target address is generated as a re-fetch address. Since the access to the GR is unnecessary, the address calculation is always performed in the BD stage. Further, among instructions that are the BTB registration type of branch instructions, but are not yet registered to the BTB (in other words, instructions that are predicted as 'not-taken'), their branch target address is registered to the BTB, respectively, regardless of the branch result (TAKEN or NOT-TAKEN) (the cases ②, ④, ⑥ and ⑧ in Fig. 4).

(3) Pipeline restoration processing

When the branch prediction is incorrect, an instruction(s) fetched because of the wrong prediction, and its execution result(s) are quashed (pipeline flash); simultaneously, a correct instruction flow is re-fetched (RIF: Re-Instruction Fetch) using a re-fetch address (the cases ③, ④, ⑦, ⑧ and ⑨).

5.3 Branch penalties

When it is assumed that there is no flow dependence relative to a source operand of a branch instruction, the disarray of the pipeline, which results from the branch instruction (branch penalties), will be as follows:

(a) When the branch prediction is correct (the cases ①, ②, ⑤, ⑥ and ⑩ in Fig. 4): There is no penalty.

(b) When [the addressing mode is] the PC relativity and the branch prediction is incorrect, or when 'taken' was predicted but the branch prediction is incorrect (the cases ③, ④ and ⑨ in Fig. 4): there is one cycle of penalty.

(c) When [the addressing mode is] the GR relativity or PC + GR and 'not-taken' was predicted but the branch prediction is incorrect (the cases ⑦ and ⑧ in Fig. 4): There are two cycles of penalties.

Concerning control dependence, if there is no flow dependence of the TFR, everything is resolved in the BD stage.

6. Conclusion

As explained above, the branch architecture and the branch pipeline processing in the DSNS processor were described.

In order to reduce the influences of the branch penalties, the following techniques have been adopted:

① Static branch prediction with a branch-target-buffer;
② Speculative instruction execution;
③ Advanced conditioning; and
④ Early branch resolution.

At present, hardware development is in progress. At the same time, a software simulator is being developed. The above-mentioned adopted four techniques are closely to related each other, so we are planning to evaluate interactions among these techniques in the future.

Further, this branch architecture is premised on the existence of an optimizing compiler. Concerning the static branch prediction, the boosting and the advanced conditioning, their efficacies cannot be anticipated without the optimizing compiler. Especially, in order to make the boosting and the advanced conditioning be effective, the development of a high degree of static code scheduling algorithm is essential. We would like to report about these optimizing compiler technologies in a separate opportunity.

References:

[1] Hara et al.: "Instruction Supply Mechanism of Superscalar Processor 'Shinpu' Based upon SIMP (single instruction flow/ multiplexed instruction pipeline) Method", Research Paper of Information Processing Society of Japan, ARC-80-7 (January 1990)

[2] Hara et al.: "Conditional Branch Method for 'Shinpu' Processor", The 40th Convention of Information Processing Society of Japan, 7L6 (March 1990)

[3] Murakami et al.: "Policy for Improvement of Superscalar Processor Based upon SIMP (Single instruction flow/ Multiplexed instruction pipeline) Type Method", Technical Report of The Institute of Electronics Information and Communication Engineers, CPSY-90-54 (July 1990)

[4] Hara et al.: "Configuration and Processing of Improved Version of Superscalar Processor Based upon SIMP (Single instruction flow/ Multiplexed instruction pipeline) Method", Technical Report of The Institute of Electronics Information and Communication Engineers, CPSY-90-55 (July 1990)

[5] Notomi [or Iritomi or Itomi] et al.: "Load/store Pipeline of DSN Superscalar Processor Prototype", Research Paper of Information Processing Society of Japan, ARC-86-4 (January 1991)

[6] Ditzel, D. R. and McLellan, H. R.: "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proc. 14th Ann. Int'l Symp. Computer Architecture, pp. 2 – 9, June 1987

[7] Hwu, W. W. and Patt, Y. N.: "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," IEEE Trans. Comput., Vol. C-36, No. 12, pp. 1496 – 1514, Dec. 1987

[8] Lee, J. K. F. and Smith, A. J.: "Branch Prediction Strategies and Branch Target Buffer Design," Computer Vol. 17, No. 1, pp. 6 – 22, Jan. 1984

[9] Lilja, D. J.: "Reducing the Branch Penalty in Pipelined Processors," Computer, Vol. 21, No. 7, pp. 47 – 55, July 1988

[10] McFarling, S. and Hennessy J.: "Reducing the Cost of Branches," Proc. 13th Ann. Int'l. Symp. Computer Architecture, pp. 396 – 403, June 1986

[11] Pleszkun, A. R. et al.: "WISQ: A Restartable Architecture Using Queues," Proc. 14th Ann. Int'l. Symp. Computer Architecture, pp. 290 – 299, June 1987

[12] Rosocha, W. G. and Lee, E. S.: "Performance Enhancement of SISD Processors," Proc. 6th Ann. Symp. Computer Architecture, pp. 216 – 231, Apr. 1979

[13] Smith, J. E.: "A Study of Branch Prediction Strategies," Proc. 8th Ann. Symp. Computer Architecture, pp. 135 – 148, 1981

[14] Smith, J. E. and Pleszkun, A. R.: "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans. Comput., Vol. 37, No. 5, pp. 562 – 573, May 1988

[15] Smith, M. D., Lam, M. S. and Horowitz, M. A.: "Boosting Beyond Static Scheduling in a Superscalar Processor," Proc. 17th Ann. Int'l Simp. Computer Architecture, pp. 344 – 354, May 1990

[16] Sohi, G. S. and Vajapeyam, S.: "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," Proc. 14th Ann. Int'l. Symp. Computer Architecture, pp. 27 – 34, June 1987

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.